CS76 22F: PA3 Chess - Genghe Zhu

Description

Minimax AI and Cutoff Test

Minimax AI works by implementing the pseudo code in the slides. It works as a sort of recursive function. The base case is determined by the `cutoff_test` function, which stops Minimax when either a certain depth is reached, or the game is won/lost/drawn. If the base case is not true, it then checks the boolean `max` to see if the function should maximize or minimize. If it is maximizing, it will call a `max_value` function which loops through the next moves and returns the move that will generate the max value with the evaluation function, and vice versa for `min_value`.

Evaluation Function

I created two evaluation functions. The first is called `material_evaluation`, which just adds up the piece totals. This works by subtracting one side's totals form another side's. The second is called `positional_evaluation`, which adds piece totals and multiplies them by a position (relative to the middle of the board). It does so with this line: `own_piece_tot += self.piece_value(piece) * (pos_imp - (0.1 * abs(row - 4.5)) - (0.1 * abs(col - 4.5)))`, where `pos_imp` is a hyperparameter I tuned to be 2, `row` is the row number of the piece, and `col` is the column number of the piece. This makes it so that pieces in (4,4), (4,5), (5,4), or (5,5) have the best positions. The evaluation functions both return a huge number that outweighs any sums if checkmate can be reached.

The `choose_move` function then calls the minimax function, which calls the evaluation functions. It starts with the maximizing function and then calls the minimizing function.

Alpha-Beta Pruning

Alpha Beta AI works very similarly to Minimax AI, but just passes in extra alpha and beta values. If it is maximizing, alpha will be compared with the current best value, and will be equal the the greater of the two. Then it will prune if beta is less than or equal to alpha. If it is minimizing, beta will be equal to the smaller value between the current beta and the best value. If beta is less than or equal to alpha, it will prune. The rest is the same as Minimax AI.

Iterative Deepening

Iterative deepening works by calling `choose_move` in Alpha Beta AI for each depth from 1 to max depth. Each call will result in a recommended best move for its depth. It will also print out the number of nodes visited.

Evaluation

Minimax AI

Minimax AI works pretty well and makes good moves in the mid game with the material evaluation. It works well on depths 1 to 3, but takes too long to run at greater depths. I tested this by running the code against the Random AI, and Minimax AI beats random AI every time. I also tested it by running the Minimax AI

against the chess puzzle described in EdDiscussion, which moves exactly as predicted. Minimax works better with the positional evaluation function, as it moves pieces early into the middle when no pieces can be taken. A problem I faced with this evaluation function was that in the late game, it would not promote pawns to queens since it would just have them stay in the middle.

Alpha-Beta AI

Alpha beta pruning worked well since it had the exact same results as minimax, but in much less nodes visited. I verified this by running it against a lower number depth alpha beta AI. I.e. alpha beta depth 3 vs alpha beta depth 1 plays the same moves as minimax depth 3 vs minimax depth 1. It looked like it was making intelligent moves and also passed the puzzle.

Iterative Deepening AI

Iterative deepening showed at different depths, minimax and alpha beta returned different moves based off what is optimal foe each depth. As we go deeper, it is less greedy and focuses on maximizing long term value.

Responses to discussion questions

minimax and cutoff test

The minimax algorithm works well up to a depth of 3. Anything above a depth of 3 would take too long as it would visit too many nodes and take forever to choose a move every turn. A typical call with a depth of 3 would run from around 5000 to 25000 nodes visited every move.

evaluation function

I used two main evaluation functions. The first just looked at the materials on the board. This means that it would look at the differences between pieces on both sides (pawn 1, knight 3, bishop 3, rook 5, queen 9). This only worked well in the mid game where pieces could actually be exchanged. I thus also implemented a positional evaluation function which takes into account the piece's location on the board. This prioritizes putting pieces in the middle, which helps the algorithm move pieces in the beginning when no pieces can be taken.

alpha-beta

Alpha beta move reordering made sure that the same move wasn't played every time. This shuffle helped so that different moves would be played when no pieces could be taken with the material evalutation.

iterative deepening

I verified that at different depths, best_move changes and improves. The algorithm becomes less greedy and looks for long-term value.